# 模块 Module

## 定义

包含一系列数据、函数、类的文件，通常以.py结尾。

## 作用

让一些相关的数据，函数，类有逻辑的组织在一起，使逻辑结构更加清晰。

有利于多人合作开发。

## 导入

### import

1. 语法：

import 模块名

import 模块名 as 别名

1. 作用：将某模块整体导入到当前模块中
2. 使用：模块名.成员

### from import

1. 语法：

from 模块名 import 成员名[ as 别名1] (成员：函数和变量)

作用：将模块内的一个或多个成员导入到当前模块的作用域中。

### from import \*

1. 语法：from 模块名 import \*
2. 作用：将某模块的所有成员导入到当前模块。
3. 模块中以下划线(\_)开头的属性，不会被导入，通常称这些成员为隐藏成员。

**Demon01:****模块****def** fun01():  
 print(**"demo01 - fun01"**)  
fun01()  
**# 导入方式1:import 模块名  
# 本质：创建变量名，关联该模块。****import** module01  
module01.fun02()  
module01.fun01()  
**# 导入模块时，可以起一个别名**# import module01 as m  
# m.fun02()**# 导入方式2:from 模块 import 成员  
# 本质：将该模块成员导入到当前作用域中**# from module01 import fun02  
# fun02()**# 导入方法3: from 模块 import \*  
# 小心：导入的成员命名冲突**# from module01 import \*  
# fun02()  
# fun03()  
# fun01()# 调用的不是自己的，而是module01的。

**Demo02: 包**

"""  
项目根目录--普通文件夹  
包--文件夹  
 模块--文件  
 类--class  
 函数--def  
 语句--命令  
**# 导入模块成功的唯一条件：  
# sys.path + 导入路径 可以正确定位模块**  
import sys  
sys.path.append('/home/tarena/month01/day15/my\_project')  
"""  
**# 导入方式1**  
# import package01.p1  
# package01.p1.fun01()  
  
# import package01.p1 as p  
# p.fun01()  
**# 导入方式2**  
# from package01.package02.p2 import fun02  
# fun02()  
**# 导入方式3**  
# from package01.package02.p2 import \*  
# fun02()  
**# 如果导入路径是一个包，那么需要在包的\_\_init\_\_.py模块中设置\_\_all\_\_属性**  
**from** package01.package02 **import** \*  
p2.fun02()

**模块变量**

1. \_\_all\_\_变量：定义可导出成员，仅对from xx import \*语句有效。
2. \_\_doc\_\_变量：文档字符串。（很少使用）
3. \_\_file\_\_变量：模块对应的文件路径名。（很少使用）
4. \_\_name\_\_变量：模块自身名字，可以判断是否为主模块。

当此模块作为主模块(第一个运行的模块)运行时，\_\_name\_\_绑定'\_\_main\_\_'，不是主模块，而是被其它模块导入时,存储模块名。

**Demo03:**

"""  
 模块相关知识点  
"""  
**# 1. \_\_all\_\_变量：定义可导出成员，仅对from xx import \*语句有效。**  
# from module02 import \*  
# fun01()  
# MyClass.fun03()  
**from** module02 **import** MyClass  
# MyClass.fun03()  
**# 2. 模块对应的文件路径名。**  
# print(\_\_file\_\_)  
**# 3.\_\_name\_\_：真实的模块名（被导入模块） \_\_main\_\_（主模块）**  
print(\_\_name\_\_)# \_\_main\_\_

## 加载过程

在模块导入时，模块的所有语句会执行。

如果一个模块已经导入，则再次导入时不会重新执行模块内的语句。

## 分类

1. 内置模块(builtins)，在解析器的内部可以直接使用。
2. 标准库模块，安装Python时已安装且可直接使用。
3. 第三方模块（通常为开源），需要自己安装。
4. 用户自己编写的模块（可以作为其他人的第三方模块）

**Demo04:**

*"""* 标准库模块 *time 时间  
 练习:exercise02/03  
"""***import** time  
  
**# 1. 当前时间戳（1970年1月1日到现在经过的秒数）**print(time.time()) *# 1574153197.3360288***# 2. 时间元组(年,月,日,时,分,秒,星期,年的天,夏令时偏移量)**tuple\_time = time.localtime()  
print(tuple\_time)  
print(tuple\_time[1])  
print(tuple\_time[3:6])  
**# 3. 时间戳 --> 时间元组**print(time.localtime(1153197.3360288))  
**# 4. 时间元组-->时间戳**print(time.mktime(tuple\_time))  
**# 5. 时间元组--> 字符串**print(time.strftime(**"%Y年%m月%d日 %H小时%M分钟%S秒"**, tuple\_time))  
**# 6.字符串 --> 时间元组**print(time.strptime(**"2019-11-19 17:10:13"**, **"%Y-%m-%d %H:%M:%S"**))

## 搜索顺序

搜索内建模块(builtins)

sys.path 提供的路径，通常第一个是程序运行时的路径。

# 包package

## 定义

将模块以文件夹的形式进行分组管理。

## 作用

让一些相关的模块组织在一起，使逻辑结构更加清晰。

## 导入

from 包名 import 模块名 [as 模块新名]

from 包名.子包名 import 模块名 [as 模块新名]

from 包名.子包名.模块名 import 成员名 [as 属性新名]

# 导入包内的所有子包和模块

from 包名 import \*

from 包名.模块名 import \*

import 包名 [as 包别名] 需要设置\_\_all\_\_

import 包名.模块名 [as 模块新名]

import 包名.子包名.模块名 [as 模块新名]

## 搜索顺序

sys.path 提供的路径

## \_\_init\_\_.py 文件

是包内必须存在的文件

会在包加载时被自动调用

### \_\_all\_\_

记录from 包 import \* 语句需要导入的模块

案例：

my\_ project /

\_\_init\_\_.py

main.py

common/

\_\_init\_\_.py

double\_list\_helper.py

list\_helper.py

skill\_system/

\_\_init\_\_.py

skill\_deployer.py

skill\_manager.py

# 异常处理Error

## 异常

1. 定义：运行时检测到的错误。
2. 现象：当异常发生时，程序不会再向下执行，而转到函数的调用语句。
3. 常见异常类型：

-- 名称异常(NameError)：变量未定义。

-- 类型异常(TypeError)：不同类型数据进行运算。

-- 索引异常(IndexError)：超出索引范围。

-- 属性异常(AttributeError)：对象没有对应名称的属性。

-- 键异常(KeyError)：没有对应名称的键。

-- 为实现异常(NotImplementedError)：尚未实现的方法。

-- 异常基类Exception。

## 处理

1. 语法：

try:

可能触发异常的语句

except 错误类型1 [as 变量1]：

处理语句1

except 错误类型2 [as 变量2]：

处理语句2

except Exception [as 变量3]：

不是以上错误类型的处理语句

else:

未发生异常的语句

finally:

无论是否发生异常的语句

1. 作用：将程序由异常状态转为正常流程。
2. 说明：

as 子句是用于绑定错误对象的变量，可以省略

except子句可以有一个或多个，用来捕获某种类型的错误。

else子句最多只能有一个。

finally子句最多只能有一个，如果没有except子句，必须存在。

如果异常没有被捕获到，会向上层(调用处)继续传递，直到程序终止运行。

**Demo01：**

*"""* **异常处理** *"""  
# 常见错误类型  
# print(qtx)  
# print("结果是："+10)  
# print([1,2,3][10])  
  
# class MyClass:  
# pass  
# c01 = MyClass()  
# print(c01.qtx)  
  
# print({"a":1}["b"])***def** div\_apple(apple\_count):  
 person\_count = int(input(**"请输入人数："**)) *# ValueError* result = apple\_count / person\_count *# ZeroDivisionError* print(**"每个人%f个苹果"** % result)  
  
  
**# 1. 统一处理所有异常****try**:  
 div\_apple(10)  
**except**:  
 print(**"出错啦"**)  
  
**# 2.针对不同错误，做出相应的处理逻辑** *# try:  
# div\_apple(10)  
# except ValueError:  
# print("输入的不是整数")  
# except ZeroDivisionError:  
# print("0不能作为分母")***# 3. 没有错误的逻辑 + 出错的逻辑** *# try:  
# div\_apple(10)  
# except ValueError:  
# print("输入的不是整数")  
# except ZeroDivisionError:  
# print("0不能作为分母")  
# else:  
# print("没有错误的逻辑")* **# 4.出错但是解决不了，可是具有必须执行的逻辑。** *# try:  
# div\_apple(10)  
# finally:  
# print("无论是否异常，都要执行的逻辑")*print(**"后续逻辑"**)

## raise 语句

1. 作用：抛出一个错误，让程序进入异常状态。
2. 目的：在程序调用层数较深时，向主调函数传递错误信息要层层return 比较麻烦，所以人为抛出异常，可以直接传递错误信息。。

**Demo02：**

"""  
  **主动抛出异常 --> 快速传递错误信息  
 自定义异常类 --> 封装数据**  
"""  
  
  
**class** AgeError(Exception):  
 **def** \_\_init\_\_(self, message=**""**, code=**""**, id=0):  
 self.message = message  
 self.code = code  
 self.id = id  
  
  
**class** Wife:  
 **def** \_\_init\_\_(self, age=0):  
 self.age = age  
  
 @property  
 **def** age(self):  
 **return** self.\_\_age  
  
 @age.setter  
 **def** age(self, value):  
 **if** 22 <= value <= 65:  
 self.\_\_age = value  
 **else**:  
 # raise AgeError("年龄超过范围了", "if 22 <= value <=65", 1001)  
 **raise** Exception(**"年龄超过范围了"**, **"if 22 <= value <=65"**, 1001)  
  
  
**try**:  
 w01 = Wife(550)  
# 给异常对象起一个别名  
# except AgeError as e:  
# print(e.code)  
# print(e.id)  
# print(e.message)  
**except** Exception **as** e:  
 print(e.args[0])  
 print(e.args[1])  
 print(e.args[2])

## 自定义异常(抛和接两步走-先抛后接)

1. 定义：

class 类名Error(Exception):

def \_\_init\_\_(self,参数):

super().\_\_init\_\_(参数)

self.数据 = 参数

1. 调用：

try:

….

raise 自定义异常类名(参数)

….

except 定义异常类 as 变量名:

变量名.数据

1. 作用：封装错误信息

# 迭代

每一次对过程的重复称为一次“迭代”，而每一次迭代得到的结果会作为下一次迭代的初始值。例如：循环获取容器中的元素。

## 可迭代对象iterable

1. 定义：具有\_\_iter\_\_函数的对象，可以返回迭代器对象。
2. 语法

-- 创建：

class 可迭代对象名称:

  def \_\_iter\_\_(self):

      return 迭代器

-- 使用：

for 变量名 in 可迭代对象:

语句

1. 原理：

迭代器 = 可迭代对象.\_\_iter\_\_()

while True:

try:

print(迭代器.\_\_next\_\_())

except StopIteration:

break

## 迭代器对象iterator

1. 定义：可以被next()函数调用并返回下一个值的对象。
2. 语法

class 迭代器类名:

def \_\_init\_\_(self, 聚合对象):

self.聚合对象= 聚合对象

def \_\_next\_\_(self):

if 没有元素:

raise StopIteration

return 聚合对象元素

1. 说明：

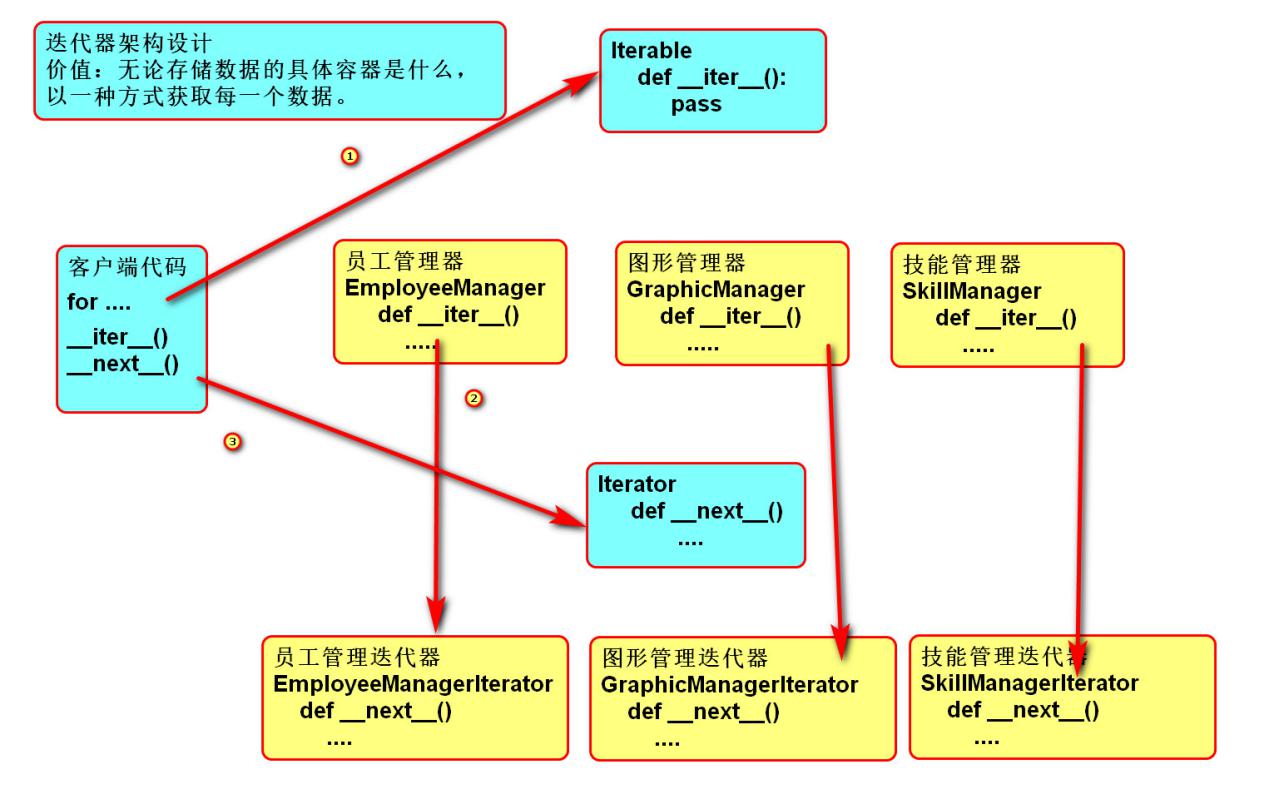
-- 聚合对象通常是容器对象。

1. 作用：使用者只需通过一种方式，便可简洁明了的获取聚合对象中各个元素，而又无需了解其内部结构。

**Demo03：**

"""  
 **可迭代对象**  
"""  
list01 = [4, 4, 54, 56, 76]  
# 迭代  
**for** item **in** list01:  
 print(item)  
  
# 笔试题：  
**# 可以被for的条件：**  
**# 对象具有\_\_iter\_\_()方法**  
**# for循环原理:**

**获取迭代器-->获取下一个元素--> 如果没有元素则结束循环**  
**# 1. 获取迭代器**  
iterator = list01.\_\_iter\_\_()  
**while True**:  
 **try**:  
**# 2. 获取下一个元素**  
 item = iterator.\_\_next\_\_()  
 print(item)  
**# 3. 如果没有元素则结束循环**  
 **except** StopIteration:  
 **break**



**Demo04：**

"""  
  **迭代器实例**  
"""  
  
  
**class** SkillManagerIterator:  
 """  
 迭代器  
 """  
  
 **def** \_\_init\_\_(self, data):  
 self.\_\_data = data  
 self.\_\_index = 0  
  
 **def** \_\_next\_\_(self):  
 # try:  
 # item = self.\_\_data[self.\_\_index]  
 # self.\_\_index += 1  
 # return item  
 # except:  
 # raise StopIteration()  
  
 **if** self.\_\_index == len(self.\_\_data):  
 **raise** StopIteration()  
  
 item = self.\_\_data[self.\_\_index]  
 self.\_\_index += 1  
 **return** item  
  
  
**class** SkillManager:  
 """  
 可迭代对象  
 """  
  
 **def** \_\_init\_\_(self):  
 self.\_\_list\_skills = []  
  
 **def** add\_skill(self, skill):  
 self.\_\_list\_skills.append(skill)  
  
 **def** \_\_iter\_\_(self):  
 **return** SkillManagerIterator(self.\_\_list\_skills)  
  
  
manager = SkillManager()  
manager.add\_skill(**"九阳神功"**)  
manager.add\_skill(**"九阴白骨爪"**)  
manager.add\_skill(**"乾坤大挪移"**)  
  
**for** item **in** manager:  
 print(item)

Exercise:

"""  
 练习:  
 定义员工管理器，记录多个员工。  
 迭代员工管理器，获取多个员工。  
 要求：体会推导过程  
 画出迭代器架构设计图  
"""  
**class** EmployeeManagerIterator:  
 **def** \_\_init\_\_(self, data):  
 self.\_\_data = data  
 self.\_\_index = 0  
  
 **def** \_\_next\_\_(self):  
 **if** self.\_\_index == len(self.\_\_data): #等于3的时候  
 **raise** StopIteration()  
  
 item = self.\_\_data[self.\_\_index]  
 self.\_\_index += 1  
 **return** item  
  
**class** EmployeeManager:  
  
 **def** \_\_init\_\_(self):  
 self.\_\_list\_employees = []  
  
 **def** add\_employee(self, skill):  
 self.\_\_list\_employees.append(skill)  
  
 **def** \_\_iter\_\_(self):  
 iterator = EmployeeManagerIterator(self.\_\_list\_employees)  
 **return** iterator  
  
manager = EmployeeManager()  
manager.add\_employee(**"悟空"**)  
manager.add\_employee(**"八戒"**)  
manager.add\_employee(**"唐僧"**)  
  
**for** item **in** manager:  
 print(item)  
**# 1.首先获取一个迭代器**  
# iterator = manager.\_\_iter\_\_()=EmployeeManager().\_\_iter\_\_()  
# while True:  
# try:  
**# 2. 获取下一个元素**  
# item = iterator.\_\_next\_\_()  
# print(item)  
**# 3. 如果没有元素则结束循环**  
# except StopIteration:  
# break

# 生成器generator

1. 定义：能够动态(循环一次计算一次返回一次)提供数据的可迭代对象。
2. 作用：在循环过程中，按照某种算法推算数据，不必创建容器存储完整的结果，从而节省内存空间。数据量越大，优势越明显。
3. 以上作用也称之为延迟操作或惰性操作，通俗的讲就是在需要的时候才计算结果，而不是一次构建出所有结果。

举个栗子：

**def** my\_range(stop):  
 begin = 0  
 **while** begin < stop:  
 **yield** begin*# 暂时离开，再次调用继续执行* begin += 1  
**for** item **in** my\_range(6):  
 print(item) *# 0 1 2 3 4*

## 生成器函数

1. 定义：含有yield语句的函数，返回值为生成器对象。
2. 语法

-- 创建：

def 函数名():

…

yield 数据

…

-- 调用：

for 变量名 in 函数名():

语句

1. 说明：

-- 调用生成器函数将返回一个生成器对象，不执行函数体。

-- yield翻译为”产生”或”生成”

1. 执行过程：
2. 调用生成器函数会自动创建迭代器对象。
3. 调用迭代器对象的\_\_next\_\_()方法时才执行生成器函数。
4. 每次执行到yield语句时返回数据，暂时离开。
5. 待下次调用\_\_next\_\_()方法时继续从离开处继续执行。
6. 原理：生成迭代器对象的大致规则如下

-- 将yield关键字以前的代码放在next方法中。

-- 将yield关键字后面的数据作为next方法的返回值。

## 内置生成器

### 枚举函数enumerate

1. 语法：

for 变量 in enumerate(可迭代对象):

语句

for 索引, 元素in enumerate(可迭代对象):

语句

1. 作用：遍历可迭代对象时，可以将索引与元素组合为一个元组。

### zip

1. 语法：

for item in zip(可迭代对象1, 可迭代对象2….):

语句

1. 作用：将多个可迭代对象中对应的元素组合成一个个元组，生成的元组个数由最小的可迭代对象决定。

## 生成器表达式

1. 定义：用推导式形式创建生成器对象。
2. 语法：变量 = ( 表达式 for 变量 in 可迭代对象 [if 真值表达式] )

Demo01：

"""  
 yield  
 练习:exercise01  
"""  
**class** MyRange:  
 """  
 可迭代对象  
 """  
 **def** \_\_init\_\_(self, stop):  
 self.\_\_stop = stop  
  
 # 程序执行过程：  
 # 调用不执行  
 # 调用\_\_next\_\_才执行  
 # 到yield暂时离开  
 # 再次调用\_\_next\_\_继续执行  
 # ...  
 # 程序执行原理： "你看见的代码，实际不是这个样子"  
 # 将yield关键字以前的代码定义到next方法中  
 # 将yield关键字以后的数据作为next方法返回值  
 # def \_\_iter\_\_(self):  
 # print("准备数据")  
 # yield 0  
 #  
 # print("准备数据")  
 # yield 1  
 #  
 # print("准备数据")  
 # yield 2  
 #  
 # print("准备数据")  
 # yield 3  
 #  
 # print("准备数据")  
 # yield 4  
  
 **def** \_\_iter\_\_(self):  
 begin = 0  
 **while** begin < self.\_\_stop:  
 **yield** begin# 暂时离开，再次调用继续执行  
 begin += 1  
  
mr = MyRange(10)  
iterator = mr.\_\_iter\_\_()  
**while True**:  
 **try**:  
 # 2. 获取下一个元素  
 item = iterator.\_\_next\_\_()  
 print(item)  
 # 3. 如果没有元素则结束循环  
 **except** StopIteration:  
 **break**# for item in MyRange(5):  
# print(item) # 0 1 2 3 4  
  
# 循环一次 计算一个 返回一个  
# for item in MyRange(99999999999999999999999999999999999999999):  
# print(item)

Demo02：

"""  
 迭代器 --> yield --> 生成器  
 练习:exercise02~04  
"""  
  
**""" 伪代码  
class Generator:   
 # 生成器 --> 迭代器 + 可迭代对象  
 def \_\_iter\_\_(self):  
 return self  
   
 def \_\_next\_\_(self):  
 ...   
"""  
  
def** my\_range(stop):  
 begin = 0  
 **while** begin < stop:  
 **yield** begin# 暂时离开，再次调用继续执行  
 begin += 1  
  
**for** item **in** my\_range(99999999999):  
 print(item) # 0 1 2 3 4  
  
# mr = my\_range(99999999999)  
# iterator = mr.\_\_iter\_\_()  
# while True:  
# try:  
# # 2. 获取下一个元素  
# item = iterator.\_\_next\_\_()  
# print(item)  
# # 3. 如果没有元素则结束循环  
# except StopIteration:  
# break

demo03：

*"""  
 生成器表达式  
 练习:exercise05  
"""*list01 = [43, 4, 54, 5, 66, 77]  
  
*# 将list01中所有元素增加10之后，存入list02  
# list02 = []  
# for item in list01:  
# list02.append(item + 10)*list02 = [item + 10 **for** item **in** list01]  
*# for item in list02:  
# print(item)  
# -------------------------------------------------  
# -------------------------------------------------  
# def get\_elements():  
# for item in list01:  
# yield item + 10  
  
# generate01 = (item + 10 for item in list01)  
# for item in generate01:  
# print(item)***for** item **in** (item + 10 **for** item **in** list01 **if** item % 2):  
 print(item)

Exercise06：

"""  
 创建老婆(身高,姓名,年龄,体重)列表  
 -- 定义函数，在老婆列表中查找身高在160 -- 170之间的所有老婆  
 -- 定义函数，在老婆列表中查找年龄25 -- 30之间的所有老婆姓名与年龄  
 使用生成器函数与生成器表达式  
"""  
# 由多个数据所描述的一个实物 --> 封装数据  
**class** Wife:  
 """  
 抽象的数据  
 """  
 **def** \_\_init\_\_(self, name=**""**, age=0, height=0, weight=0):  
 self.name = name  
 self.age = age  
 self.height = height  
 self.weight = weight  
  
 **def** \_\_str\_\_(self):  
 **return "%s--%d--%d--%d"** % (self.name, self.age, self.height, self.weight)  
  
  
**def** find01(list\_target):  
 **for** item **in** list\_target:  
 **if** 160 <= item.height <= 170:  
 **yield** item  
  
  
**def** find02(list\_target):  
 **for** item **in** list\_target:  
 **if** 25 <= item.age <= 30:  
 **yield** (item.name, item.age)  
  
# ------------------------------  
# 测试  
list01 = [  
 Wife(**"铁锤"**, 27, 190, 200),  
 Wife(**"铁钉"**, 37, 165, 160),  
 Wife(**"铁棒"**, 24, 160, 190),  
 Wife(**"铁锅"**, 23, 190, 100),  
]  
  
**for** item **in** find01(list01):  
 print(item)  
  
**for** item **in** (item **for** item **in** list01 **if** 160 <= item.height <= 170):  
 print(item)  
  
**for** item **in** ((item.name, item.age) **for** item **in** list01 **if** 25 <= item.age <= 30):  
 print(item)  
  
**1.惰性操作/延迟操作  
 优点：节省内存  
 缺点：不同灵活(索引、切片、一次)访问**  
**2.立即操作  
优点：灵活(索引、切片)访问  
缺点：占用内存过多**  
  
**# 惰性操作 --> 立即操作**  
result = tuple(find01(list01))  
print(result[-1])  
# -- 定义函数，在老婆列表中查找"铁钉"老婆对象  
**def** find03(list\_target):  
 **for** item **in** list\_target:  
 **if** item.name == **"铁钉"**:  
 **return** item  
  
result = find03(list01)  
print(result)  
# -- 定义函数，在老婆列表中查找体重在40- - 60之间的所有老婆  
**def** find04(list\_target):  
 **for** item **in** list\_target:  
 **if** 40 <= item.weight <= 60:  
 **yield** item  
**# 惰性操作**  
**for** item **in** find04(list01):  
 print(item) **# 立即操作**  
result = list(find04(list01))  
print(result[0])

Exercise07：

# 练习:  
# 定义函数，在老婆列表中查找年龄大于25的所有老婆对象  
# 定义函数，在老婆列表中查找年身高小于180的所有老婆对象  
**class** Wife:  
 """  
 抽象的数据  
 """  
  
 **def** \_\_init\_\_(self, name=**""**, age=0, height=0, weight=0):  
 self.name = name  
 self.age = age  
 self.height = height  
 self.weight = weight  
  
 **def** \_\_str\_\_(self):  
 **return "%s--%d--%d--%d"** % (self.name, self.age, self.height, self.weight)  
  
  
list01 = [  
 Wife(**"铁锤"**, 27, 190, 200),  
 Wife(**"铁钉"**, 37, 165, 160),  
 Wife(**"铁棒"**, 24, 160, 190),  
 Wife(**"铁锅"**, 23, 190, 100),  
]  
  
  
**def** find01(list\_target):  
 **for** item **in** list\_target:  
 **if** item.age > 25:  
 **yield** item  
  
  
**def** find02(list\_target):  
 **for** item **in** list\_target:  
 **if** item.height < 180:  
 **yield** item  
  
  
**def** condition01(item):  
 **return** item.age > 25  
  
  
**def** condition02(item):  
 **return** item.height < 180  
  
**# find 第一个参数：数据  
# find 第二个参数：逻辑**  
**# 万能：数据以及对数据的核心操作，都是灵活的。**  
**def** find(list\_target, func):  
 **for** item **in** list\_target:  
 # if item.height < 180:  
 # if condition02(item):  
 **if** func(item):  
 **yield** item  
  
find(list01,condition02)

# 函数式编程

1. 定义：用一系列函数解决问题。

-- 函数可以赋值给变量，赋值后变量绑定函数。

-- 允许将函数作为参数传入另一个函数。

-- 允许函数返回一个函数。

1. 高阶函数：将函数作为参数或返回值的函数。

Demo04：

*"""  
 函数式编程 -- 语法  
 回顾面向对象编程：封装(分)、继承(隔)、多态(做)  
"""***def** fun01():  
 print(**"fun01执行喽"**)  
  
*# 理论支柱1:将函数赋值给变量*a = fun01  
a()  
  
**def** fun02():  
 print(**"fun02执行喽"**)  
  
*# 理论支柱2:允许将函数作为参数传入另一个函数***def** fun03(func):  
 print(**"fun03执行喽"**)  
 func()  
  
fun03(fun01)  
fun03(fun02)

Demo05：

"""  
 **函数式编程 -- 应用**"""  
list01 = [3, 4, **"a"**, 6, **"b"**, 8, 98, ]  
# 需求1:定义函数,获取列表中所有字符串  
**def** find01():  
 **for** item **in** list01:  
 **if** type(item) == str:  
 **yield** item  
# 需求2:定义函数,获取列表中所有大于10的整数  
**def** find02():  
 **for** item **in** list01:  
 **if** type(item) == int **and** item > 10:  
 **yield** item  
# **"封装" -- 分**  
# -- **提取变化**  
**def** condition01(item):  
 **return** type(item) == str  
  
**def** condition02(item):  
 **return** type(item) == int **and** item > 10  
# -- **提取共性**  
# **"继承" -- 隔(抽象->统一)**  
# **万能查找函数**  
**def** find(target, func):  
 **for** item **in** target:  
 **if** func(item):  
 **yield** item  
# **"多态" -- 做(重写)**  
**for** item **in** find(list01, condition01):  
 print(item)

## 函数作为参数

将核心逻辑传入方法体，使该方法的适用性更广，体现了面向对象的开闭原则。

### lambda 表达式

1. 定义：是一种匿名方法。
2. 作用：作为参数传递时语法简洁，优雅，代码可读性强。

随时创建和销毁，减少程序耦合度。

1. 语法

-- 定义：

变量 = lambda 形参: 方法体

-- 调用：

变量(实参)

1. 说明：

-- 形参没有可以不填

-- 方法体只能有一条语句，且不支持赋值语句。

Demo01：

*"""  
 lambda 匿名函数：  
 语法：lambda 参数:函数体  
 价值：作为实参传递给函数  
"""  
# 有参数 有返回值  
# def fun01(a):  
# return a == "ok"  
#  
#  
# re = fun01("ok")  
# print(re)  
  
# 形参 = 实参*fun01 = **lambda** a: a == **"ok"***# 函数体*re = fun01(**"ok"**)  
print(re)  
  
  
*# 有多参数 有返回值  
# def fun02(a, b, c):  
# return a == b == c*fun02 = **lambda** a, b, c: a == b == c  
re = fun02(1, 2, 3)  
print(re)  
  
  
*# 没参数 有返回值  
# def fun03():  
# return "ok"*fun03 = **lambda** :**"ok"**re = fun03()  
print(re)  
  
  
*# 没参数 没返回值  
# def fun04(a):  
# print(a)*fun04 = **lambda** a:print(a)  
fun04(**"ok"**)  
  
*# def fun05(a):  
# a[0] = 10  
#  
# list01 = [1]  
# fun05(list01)  
# print(list01)#?  
  
# lambda 表达式不能赋值  
# fun05 = lambda a:a[0] = 10***def** fun06(a):  
 print(**"a"**)  
 **return** a == **"ok"***# lambda 表达式只支持一行语句  
# fun06 =lambda a:(  
# print("a")  
# a == "ok"  
# )*

### 内置高阶函数

1. map（函数，可迭代对象）：使用可迭代对象中的每个元素调用函数，将返回值作为新可迭代对象元素；返回值为新可迭代对象。
2. filter(函数，可迭代对象)：根据条件筛选可迭代对象中的元素，返回值为新可迭代对象。
3. sorted(可迭代对象，key = 函数,reverse = bool值)：排序，返回值为排序结果。
4. max(可迭代对象，key = 函数)：根据函数获取可迭代对象的最大值。
5. min(可迭代对象，key = 函数)：根据函数获取可迭代对象的最小值。

*Demo02：*

"""  
 内置高阶函数  
"""  
**from** common.iterable\_tools **import** IterableHelper  
  
  
**class** Wife:  
 """  
 抽象的数据  
 """  
  
 **def** \_\_init\_\_(self, name=**""**, age=0, height=0, weight=0):  
 self.name = name  
 self.age = age  
 self.height = height  
 self.weight = weight  
  
 **def** \_\_str\_\_(self):  
 **return "%s--%d--%d--%d"** % (self.name, self.age, self.height, self.weight)  
  
  
list01 = [  
 Wife(**"铁锤"**, 27, 190, 200),  
 Wife(**"铁钉"**, 37, 165, 160),  
 Wife(**"铁棒"**, 24, 160, 190),  
 Wife(**"铁锅"**, 23, 190, 100),  
]  
# 需求：在老婆列表中，查找所有体重大于160的老婆对象  
# for item in IterableHelper.find\_all(list01,lambda element:element.weight > 160):  
# print(item)  
  
**# 1. filter 过滤器**  
**for** item **in** filter(**lambda** element: element.weight > 160, list01):  
 print(item)  
  
# 需求：在老婆列表中，查找所有老婆的姓名与体重  
# for item in IterableHelper.select(list01,lambda e:(e.name,e.weight)):  
# print(item)  
  
**# 2. map 映射**  
**for** item **in** map(**lambda** e: (e.name, e.weight), list01):  
 print(item)  
  
# 需求：在老婆列表中，查找最重的老婆  
# re = IterableHelper.get\_max(list01,lambda item:item.weight)  
# print(re)  
  
**# 3. 获取最大**  
re = max(list01, key=**lambda** item: item.weight)  
print(re)  
  
**# 4. 排序**  
**# 修改列表内部元素 --> 无需通过返回值传递结果**  
# IterableHelper.order\_by(list01,lambda item:item.weight)  
# for item in list01:  
# print(item)  
  
**# 返回排序后的列表**  
# for item in sorted(list01, key=lambda item: item.weight):  
# print(item)  
**for** item **in** sorted(list01, key=**lambda** item: item.weight,reverse=**True**):  
 print(item)

## 函数作为返回值

逻辑连续，当内部函数被调用时，不脱离当前的逻辑。

### 闭包

1. 三要素：

-- 必须有一个内嵌函数。

-- 内嵌函数必须引用外部函数中变量。

-- 外部函数返回值必须是内嵌函数。

1. 语法

-- 定义：

def 外部函数名(参数):

外部变量

def 内部函数名(参数):

使用外部变量

return 内部函数名

-- 调用：

变量 = 外部函数名(参数)

变量(参数)

1. 定义：在一个函数内部的函数,同时内部函数又引用了外部函数的变量。
2. 本质：闭包是将内部函数和外部函数的执行环境绑定在一起的对象。
3. 优点：内部函数可以使用外部变量。
4. 缺点：外部变量一直存在于内存中，不会在调用结束后释放，占用内存。
5. 作用：实现python装饰器。

### 函数装饰器decorators

1. 定义：在不改变原函数的调用以及内部代码情况下，为其添加新功能的函数。
2. 语法

def 函数装饰器名称(func):

def 内嵌函数(\*args, \*\*kwargs):

需要添加的新功能

return func(\*args, \*\*kwargs)

return wrapper

@ 函数装饰器名称

def 原函数名称(参数):

函数体

原函数(参数)

1. 本质：使用“@函数装饰器名称”修饰原函数，等同于创建与原函数名称相同的变量，关联内嵌函数；故调用原函数时执行内嵌函数。

原函数名称 = 函数装饰器名称（原函数名称）

1. 装饰器链：

一个函数可以被多个装饰器修饰，执行顺序为从近到远。

**Demo03：**

"""  
 Enclosing 外部嵌套作用域  
"""  
  
  
**def** fun01():  
 a = 10 # 局部变量,外部嵌套变量  
 **def** fun02():  
 b = 20 # 局部变量  
 # print(a)# 可以读取外部嵌套变量  
 **nonlocal** a # 声明外部嵌套变量  
 a = 20  
  
 fun02()  
 print(a)  
  
fun01()

Demo04：

*"""*  ***闭包*** **调用函数，在内存中开辟栈帧。  
 函数执行过后，栈帧释放，局部变量销毁。  
 外部函数执行过后，不释放栈帧(留给内部函数使用)。** *"""***def** fun01():  
 a = 100  
  
 **def** fun02():  
 print(a)  
  
 **return** fun02  
  
  
re = fun01()  
re()*# 100*

Demo05：

*"""*  ***装饰器*** *拦截  
"""  
  
def print\_func\_name(func):  
 def wrapper(\*args, \*\*kwargs):# 合  
 print(func.\_\_name\_\_)  
 return func(\*args, \*\*kwargs)# 拆  
 return wrapper  
  
  
def say\_hello():  
 print("hello")  
  
@print\_func\_name# print\_func\_name(say\_goodbye)  
def say\_goodbye(a,b):  
 print("goodbye")  
  
  
say\_hello()  
say\_goodbye(10,20)*

**Iterable\_tools:**

**class** IterableHelper:  
 """  
  **可迭代对象助手  
 微软 集成操作框架**  
  
 """  
 # 静态方法 -- 不会自动传递参数  
  **@staticmethod  
 def find\_all(iterable, func\_condition):  
 """  
 通用的查找满足条件的所有元素。  
 :param iterable:需要被搜索的可迭代对象  
 :param func\_condition:搜索的条件  
 :return:生成器类型,可以推算出满足条件的元素  
 """  
 for item in iterable:  
 if func\_condition(item):  
 yield item**  
  
 @staticmethod  
 **def** select(iterable, func\_handle):  
 """  
  **通用的筛选可迭代对象中的元素。**  
 **:param** iterable:需要被筛选的可迭代对象  
 **:param** func\_handle:筛选的处理逻辑  
 **:return**:生成器类型,可以推算出筛选的元素  
 """  
 **for** item **in** iterable:  
 **yield** func\_handle(item)  
  
 @staticmethod  
 **def sum**(iterable, func\_handle):  
 """  
  **通用的累加功能**  
 **:param** iterable: 需要累加的可迭代对象  
 **:param** func\_handle: 累加的逻辑  
 **:return**:累加结果  
 """  
 sum\_value = 0  
 **for** item **in** iterable:  
 sum\_value += func\_handle(item)  
 **return** sum\_value  
  
 @staticmethod  
 **def delete\_all**(iterable, func\_condition):  
 count = 0  
 **for** i **in** range(len(iterable) - 1, -1, -1):  
 # if 20 <= iterable[i].age <= 25:  
 **if** func\_condition(iterable[i]):  
 **del** iterable[i]  
 count += 1  
 **return** count  
 # lambda e:e.age  
 @staticmethod  
 **def get\_max**(iterable, func\_condition):  
 max\_value = iterable[0]  
 **for** i **in** range(1, len(iterable)):  
 # if max\_value.age < iterable[i].age:  
 **if** func\_condition(max\_value) < func\_condition(iterable[i]):  
 max\_value = iterable[i]  
 **return** max\_value  
  
 @staticmethod  
 **def order\_by**(iterable,func\_condition):  
 **for** r **in** range(len(iterable) - 1):  
 **for** c **in** range(r + 1, len(iterable)):  
 # if iterable[r].height > iterable[c].height:  
 **if** func\_condition(iterable[r]) > func\_condition(iterable[c]):  
 iterable[r], iterable[c] = iterable[c], iterable[r]